C++ 中结构体和类的区别

在C++中，`struct`（结构体）和`class`（类）均可用于定义封装数据和函数的用户自定义类型。尽管两者功能相似，但存在以下关键区别：

---

1. **默认访问权限**

- `struct`：成员（数据和函数）**默认为public**，可直接从外部访问。
- `class`：成员**默认为private**，需通过公有成员函数（如getter/setter）访问私有成员。

```
struct MyStruct {
    int data; // 默认为public
};

class MyClass {
    int data; // 默认为private
public:
    int getData() const { return data; }
    void setData(int val) { data = val; }
};
```

---

2. **继承时的默认访问级别**

- `struct`：作为基类时，派生类默认以**public方式继承**基类成员。
- `class`：作为基类时，派生类默认以**private方式继承**基类成员。

```
struct BaseStruct {
    int baseData;
};

struct DerivedStruct : BaseStruct {
    // baseData 在此为public
};

class BaseClass {
    int baseData;
};

class DerivedClass : BaseClass {
    // baseData 在此为private
};
```

---

3. **使用惯例**

- `struct`：通常用于**纯数据（Plain-Old-Data, POD）**结构，不含复杂行为。
- `class`：用于遵循面向对象原则的实体，既封装数据又定义行为（如继承、多态）。

注意事项

在现代C++中，`struct`和`class`的功能界限已非常模糊。两者均可定义构造函数、析构函数、成员函数和访问修饰符。选择使用哪一种通常取决于以下因素：

- **代码意图**：数据容器（用`struct`） vs 行为封装（用`class`）。
- **项目规范**：遵循团队的编码标准或历史惯例。

总结

特性	`struct`	`class`
默认访问权限	public	private
默认继承方式	public继承	private继承
典型用途	数据聚合（轻量级）	对象封装（复杂逻辑）

静态局部变量\全局变量\局部变量的区别和使用场景

在 C++ 中，**局部变量**、**全局变量**和**静态变量**在作用域、生命周期和存储位置等方面存在显著差异。以下是对它们的详细比较及使用场景：

1. 局部变量（Local Variables）

- **定义**：在函数或代码块内部声明的变量。
- **作用域**：仅在声明它们的函数或代码块内有效。
- **生命周期**：在函数调用时创建，函数返回时销毁。
- **存储位置**：通常存储在栈（stack）中。

使用场景：适用于仅在特定函数或代码块内需要使用的数据，确保数据的临时性和局部性，避免与其他部分的代码发生冲突。

示例：

```
void exampleFunction() {
    int localVar = 10; // 局部变量
    // 仅在此函数内有效
}
```

2. 全局变量（Global Variables）

- **定义**：在所有函数外部（通常在文件顶部）声明的变量。
- **作用域**：在整个源文件中有效，如果使用 `extern` 关键字，在其他源文件中也可访问。
- **生命周期**：程序执行期间始终存在，直到程序结束。
- **存储位置**：存储在数据段（data segment）中。

使用场景：适用于需要在多个函数或文件之间共享的数据。然而，过度使用全局变量可能导致代码维护困难，建议谨慎使用。

示例：

```
int globalVar = 20; // 全局变量

void functionA() {
    globalVar = 30; // 访问全局变量
}

void functionB() {
    int localVar = globalVar; // 将全局变量值赋给局部变量
}
```

3. 静态局部变量（Static Local Variables）

- **定义**：在函数内部使用 `static` 关键字声明的变量。
- **作用域**：仅在声明它们的函数内有效。
- **生命周期**：在程序执行期间始终存在，函数调用结束后其值不会丢失，下次调用该函数时，变量保持上次的值。
- **存储位置**：通常存储在数据段（data segment）中。

使用场景：适用于需要在多次函数调用之间保留状态信息的情况，例如计数函数调用次数等。

示例：

```
void counterFunction() {
    static int count = 0; // 静态局部变量
    count++;
    std::cout << "Function called " << count << " times." << std::endl;
}
```

注意：静态局部变量的初始化仅在第一次调用函数时进行一次，之后的调用将保留上次的值。

总结：

- **局部变量**：用于函数内部，生命周期仅限于函数调用期间，适用于临时数据存储。
- **全局变量**：在整个程序中可访问，适用于共享数据，但可能导致命名冲突和维护困难。
- **静态局部变量**：在函数内部，生命周期贯穿整个程序执行过程，适用于需要在多次函数调用之间保留数据的场景。

理解这些变量的特性，有助于在编程中做出合理的设计选择，确保代码的清晰性和可维护性。

C++强制类型转换关键字

在 C++ 中，类型转换对于在不同数据类型之间进行转换至关重要。该语言提供了四种主要的强制类型转换运算符来执行显式类型转换：`static_cast`、`dynamic_cast`、`reinterpret_cast` 和 `const_cast`。每种运算符都有特定的用途，应在适当的场景中使用，以确保代码的安全性和清晰度。

1. `static_cast`

用途：用于在相关类型之间进行定义明确的编译时转换。

用法：

- 在数值类型之间进行转换（例如，从 `int` 转换为 `float`）。
- 在继承层次结构中转换指针或引用（向上转换或向下转换），前提是这种关系是已知且安全的。
- 调用显式构造函数或转换运算符。

示例：

```
#include <iostream>

class Base {
public:
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void show() { std::cout << "Derived class" << std::endl; }
};

int main() {
    Base* basePtr = new Derived();
    // 向上转换：从 Derived* 到 Base*
    Derived* derivedPtr = static_cast<Derived*>(basePtr);
    derivedPtr->show(); // 输出：Derived class
    delete basePtr;
    return 0;
}
```

注意：要确保在类层次结构中进行的转换是有效的，以避免出现未定义行为。

2. `dynamic_cast`

用途：在继承层次结构中安全地转换指针或引用，会进行运行时检查以确保转换的有效性。

用法：

- 在多态层次结构中进行向下转换（从基类指针/引用转换为派生类指针/引用）。
- 基类中至少需要有一个虚函数，以支持运行时类型识别（RTTI）。

示例:

```
#include <iostream>

class Base {
public:
    virtual ~Base() = default; // 虚析构函数启用 RTTI
};

class Derived : public Base {
public:
    void show() { std::cout << "Derived class" << std::endl; }
};

int main() {
    Base* basePtr = new Derived();
    // 使用 dynamic_cast 进行向下转换
    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
    if (derivedPtr) {
        derivedPtr->show(); // 输出: Derived class
    } else {
        std::cout << "Invalid cast" << std::endl;
    }
    delete basePtr;
    return 0;
}
```

注意：如果转换无效（例如，转换为对象层次结构中不存在的类型），对于指针，`dynamic_cast` 会返回 `nullptr`；对于引用，会抛出 `std::bad_cast` 异常。

3. `reinterpret_cast`

用途：在不相关的类型之间进行低级别的按位转换。

用法:

- 在不同的指针类型之间进行转换，例如从 `int*` 转换为 `char*`。
- 在整数值和指针类型之间进行相互转换。

示例:

```
#include <iostream>

int main() {
    int num = 65;
    // 将 num 的地址重新解释为指向 char 的指针
```

```
    char* charPtr = reinterpret_cast<char*>(&num);
    std::cout << *charPtr << std::endl; // 输出: A
    return 0;
}
```

注意事项：如果使用不当，`reinterpret_cast` 可能会导致未定义行为，因为它允许在不一定兼容的类型之间进行转换。使用时要格外谨慎。

4. `const_cast`

用途：为变量添加或移除 `const` 限定符。

用法：

- 移除 `const` 限定符，以便修改作为常量引用或指针传递的变量。
- 当函数接口需要时，为非 `const` 变量添加 `const` 限定符。

示例：

```
#include <iostream>

void modify(int& value) {
    value = 100;
}

int main() {
    const int num = 10;
    // 移除常量性，以便传递给一个会修改该值的函数
    modify(const_cast<int&>(num));
    std::cout << num << std::endl; // 输出: 100
    return 0;
}
```

注意：使用 `const_cast` 移除 `const` 限定符并修改最初声明为 `const` 的变量会导致未定义行为。要确保对象最初不是常量。

总结：

- `static_cast`：用于在相关类型之间进行标准的编译时转换。
- `dynamic_cast`：用于在多态类层次结构中进行安全的向下转换，并进行运行时检查。
- `reinterpret_cast`：用于在不相关的类型之间进行低级别的按位转换；使用时要格外小心。
- `const_cast`：用于为指针或引用添加或移除 `const` 限定符。

选择合适的强制类型转换运算符可以确保 C++ 代码的类型安全和清晰度。

Difference between heap and stack in C++

在 C++ 中，内存分配主要通过两个区域来管理：**栈（stack）** 和 **堆（heap）**。理解它们之间的区别对于有效的内存管理和程序性能至关重要。

栈

- **用途**：栈用于存储局部变量和函数调用信息。
- **分配与释放**：内存分配是自动进行的；当调用一个函数时，其局部变量会被压入栈中，当函数退出时，这些变量会从栈中弹出。
- **大小限制**：栈的大小是有限的，如果超过这个限制，尤其是在深度递归或使用大型局部变量的情况下，可能会导致栈溢出。
- **访问速度**：由于栈采用后进先出（LIFO）结构，并且离 CPU 缓存较近，所以访问栈内存通常更快。

示例：

```
void function() {
    int localVar = 10; // 存储在栈上
    // 函数操作
} // localVar 在此处自动销毁
```

堆

- **用途**：堆用于动态内存分配，允许在运行时分配内存。
- **分配与释放**：必须使用 `new` 等运算符显式地分配内存，并使用 `delete` 释放内存。如果不释放内存，会导致内存泄漏。
- **大小限制**：堆通常比栈大得多，其大小受系统可用内存的限制，因此适合分配大型数据结构。
- **访问速度**：由于动态内存管理的开销，访问堆内存通常比访问栈内存慢。

示例：

```
void function() {
    int* dynamicVar = new int(10); // 分配在堆上
    // 使用 dynamicVar
    delete dynamicVar; // 释放已分配的内存
}
```

主要区别：

- **生命周期**：
  - **栈**：变量仅在函数的作用域内存在；当函数退出时，它们会自动销毁。
  - **堆**：变量会一直存在，直到被显式释放，因此可以实现动态的生命周期。
- **大小限制**：
  - **栈**：大小有限；过度使用可能会导致栈溢出。
  - **堆**：容量更大，适合分配大型或可变大小的数据结构。
- **性能**：
  - **栈**：由于其结构化的性质，分配和释放速度更快。

- **堆**：由于管理动态内存的复杂性，操作速度较慢。

使用建议：

- **栈**：适用于在编译时就已知大小的小型、短期使用的变量。
- **堆**：适用于大型、复杂的数据结构，或者在事先无法确定所需内存量的情况。

理解何时以及在何处分配内存对于编写高效、可靠的 C++ 程序至关重要。对于临时的小规模数据使用栈，对于动态的大规模数据使用堆，可以实现最佳的性能和资源管理。

C++ Memory Partitioning

1. **栈区（Stack Area）**
   - **管理方式**：由编译器自动管理。在程序执行过程中，编译器负责处理栈区的各种操作，例如函数调用时参数的传递、局部变量的创建与销毁等。
   - **存储内容**：存放局部变量，即定义在函数内部的变量；函数参数，用于在函数调用时传递数据；以及返回地址，当函数执行完毕后，程序根据返回地址返回到调用该函数的位置继续执行。
   - **增长方向**：栈区向低内存地址方向增长。当有新的局部变量或函数调用时，栈顶指针会向低地址移动，分配新的栈空间；当函数执行结束，局部变量销毁时，栈顶指针向高地址移动，释放栈空间。
2. **堆区（Heap Area）**
   - **管理方式**：由程序员手动管理。程序员需要使用特定的函数（如`malloc()`用于分配内存，`free()`用于释放内存）来控制堆区内存的使用。
   - **用途**：主要用于动态内存分配。当程序在运行时需要额外的内存空间，且这些空间的大小在编译时无法确定时，就会在堆区进行分配。例如，创建一个大小由用户输入决定的数组，就需要在堆区分配内存。
   - **增长方向**：堆区向高内存地址方向增长。随着不断地分配内存，堆区会从低地址向高地址扩展。但由于手动管理的特性，如果程序员在分配内存后没有及时释放，可能会导致内存泄漏等问题。
3. **全局（静态）区（Global/Static Area）**
   - **存储内容**：存储全局变量和静态变量。全局变量在整个程序中都可以访问，其作用域不受函数限制；静态变量在函数内部定义时，其生命周期贯穿整个程序运行过程，且只初始化一次。
   - **细分区域**：分为未初始化的静态数据区（`.bss`）和已初始化的静态数据区（`.data`）。`.bss`段用于存放未初始化的全局变量和静态变量，在程序加载时，系统会自动将这部分内存初始化为0；`.data`段用于存放已经初始化的全局变量和静态变量，这些变量的值在程序中被明确指定。
4. **常量区（Constant Area）**
   - **存储内容**：主要存储常量，如字符串字面量（例如`"Hello, World!"`）。这些常量在程序运行过程中不会被修改。
   - **所在段**：通常是`.rodata`段（只读数据段）的一部分。这意味着该区域的内容只能被读取，不能被写入，防止程序意外修改常量值，保证了程序的稳定性和安全性。
5. **代码区（Code Area）**
   - **存储内容**：存储程序的可执行代码，也就是`.text`段。这部分代码包含了程序运行时执行的指令序列，是程序功能的具体实现部分。

- **特点**：代码区是只读的，这是为了防止程序在运行过程中意外修改自身的指令，确保程序执行的准确性和稳定性。同时，代码区通常是共享的，多个进程可以共享同一份代码，提高内存的使用效率。

C++ memory leaks? How to avoid it?

为了避免 C++ 中的内存泄漏，可遵循以下策略：

1. **使用智能指针**

- `std::unique_ptr`：用于独占所有权。当它超出作用域时，会自动释放所管理的内存。
- `std::shared_ptr`：用于共享所有权。使用引用计数机制，当最后一个共享指针被销毁时，会自动删除所管理的内存。
- `std::weak_ptr`：用于打破循环引用（例如与 `shared_ptr` 配合使用时）。
- 优先使用 `std::make_unique` 和 `std::make_shared` 来安全地创建智能指针。

2. **利用 RAII（资源获取即初始化）原则**

- 将资源（内存、文件等）封装在对象中。当对象超出作用域时，其析构函数会自动释放资源。
- 示例：使用 `std::vector` 或 `std::string` 代替原生数组。

3. **避免手动内存管理**

- 尽量减少使用 `new`/`delete` 或 `malloc`/`free`。而是使用标准模板库（STL）容器（如 `std::vector`、`std::map`）和智能指针。
- 对于数组，使用 `std::unique_ptr<T[]>` 或 `std::vector` 来避免 `delete[]` 可能出现的错误。

4. **遵循三/五法则**

- 如果一个类管理资源，需要显式定义（或删除）拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符和析构函数。

5. **使用内存泄漏检测工具**

- **Valgrind**：可以检测内存泄漏和内存错误。
- **AddressSanitizer（ASan）**：使用 `-fsanitize=address`（GCC/Clang 编译器）进行编译，可在运行时识别内存泄漏。
- 集成开发环境（IDE）工具（如 Visual Studio 的调试器）。

6. **安全地处理异常**

- 使用智能指针或遵循 RAII 原则，以确保即使抛出异常，资源也能被释放。
- 示例：

```
void func() {
    auto ptr = std::make_unique<int>(42); // 即使抛出异常也会自动释放内存
    // 可能抛出异常的代码
}
```

7. **避免循环引用**

- 在 `shared_ptr` 中使用 `weak_ptr` 作为非拥有型引用，以打破循环引用。

8. **匹配内存分配和释放方式**

- 对于 `new` 使用 `delete`，对于 `new[]` 使用 `delete[]`，对于 `malloc()` 使用 `free()`。
- 绝不要混用不同的内存分配器（例如，不要将 `new` 和 `free()` 混用）。

9. **使用作用域守卫**

- 对于非内存资源（如文件、锁），使用遵循 RAII 原则的包装器，如 `std::fstream` 或带有析构函数以释放资源的自定义类。

示例代码：

```
#include <memory>
#include <vector>

void safe_function() {
    // 用于独占所有权的智能指针
    auto ptr = std::make_unique<int>(10);

    // 带有引用计数的共享所有权智能指针
    auto shared = std::make_shared<int>(20);

    // STL 容器会在内部管理动态内存
    std::vector<int> vec = {1, 2, 3};

    // 无需显式调用 delete；资源会自动释放
}
```

总结：

- 借助 RAII 原则和智能指针**自动清理资源。**
- **优先使用 STL 容器**而非原生指针。
- 使用 Valgrind/ASan 等工具**检测内存泄漏。**
- **除非绝对必要，避免手动使用 `new`/`delete`。**

通过遵循这些实践，你可以消除 C++ 中大部分的内存泄漏问题。

What is a Smart Pointer? What are the types?

C++中的**智能指针**是一个对象，它充当原始指针的包装器，提供自动内存管理功能，并有助于防止诸如内存泄漏和悬空指针之类的常见问题。智能指针确保在资源不再使用时正确释放它们，这与RAII（资源获取即初始化）原则相一致。

现代C++中有三种主要的智能指针类型：

1. `std::unique_ptr`：
   - **所有权**：独占所有权；在任何时候，一个给定的资源只能由一个`std::unique_ptr`拥有。
   - **复制和移动语义**：不能被复制，但可以被移动，将所有权转移到另一个`std::unique_ptr`。
   - **用例**：适用于资源有单一所有者的情况，并且希望在所有者超出作用域时确保资源自动被清理。
   - **示例**：

```
std::unique_ptr<int> ptr1 = std::make_unique<int>(10);
std::unique_ptr<int> ptr2 = std::move(ptr1); // 转移所有权
// ptr1现在为nullptr
```

```
- **常用用法**：在函数内部创建`std::unique_ptr`来管理局部对象，当函数返回时，对象会被自动释放。例如，用于管理动态分配的数组或单个对象，确保对象生命周期与作用域绑定。还可以作为函数返回值，安全地将对象的所有权转移出函数。
```

2. `std::shared_ptr`：
   - **所有权**：共享所有权；多个`std::shared_ptr`实例可以拥有同一个资源。
   - **引用计数**：维护一个引用计数，用于跟踪有多少个`std::shared_ptr`实例指向该资源。当最后一个拥有该资源的`std::shared_ptr`被销毁或重置时，资源会被自动释放。
   - **用例**：当一个资源需要在程序的多个部分之间共享时非常理想。
   - **示例**：

```
std::shared_ptr<int> ptr1 = std::make_shared<int>(10);
std::shared_ptr<int> ptr2 = ptr1; // ptr1和ptr2都拥有该资源
// 当ptr1和ptr2都被销毁时，资源被释放
```

```
- **常用用法**：用于在不同的函数或对象之间共享数据，比如在多个模块都需要访问同一个配置信息对象时，可以使用`std::shared_ptr`来管理该配置对象，确保在所有使用者都不再需要它时才释放内存。还常用于实现对象的观察者模式，多个观察者可以共享被观察对象的指针。
```

3. `std::weak_ptr`：
   - **所有权**：非拥有型引用；不影响资源的引用计数。
   - **用例**：用于打破`std::shared_ptr`实例之间的循环引用，防止内存泄漏。在访问资源之前，`std::weak_ptr`必须转换为`std::shared_ptr`。
   - **示例**：

```
std::shared_ptr<int> ptr1 = std::make_shared<int>(10);
std::weak_ptr<int> weakPtr = ptr1; // weakPtr不影响引用计数
```

```
if (auto sharedPtr = weakPtr.lock()) {
    // 使用sharedPtr访问资源
}
```

这些智能指针在<memory>头文件中定义，是C++11及更高版本标准的一部分。它们为原始指针提供了更安全、更方便的替代方案，自动管理内存并降低出错风险。

常用用法：在存在循环引用的场景中，比如两个对象相互引用，使用std::weak_ptr来打破循环。例如，在实现树形结构或图形结构时，节点之间可能存在相互引用，使用std::weak_ptr可以避免内存泄漏。另外，也可用于观察std::shared_ptr管理的对象，在不增加对象引用计数的情况下获取对象的临时访问权。
在 C++ 中，std::weak_ptr 是一种不控制所指向对象生命周期的智能指针，它主要用于辅助 std::shared_ptr 工作，解决循环引用问题。以下是 std::weak_ptr 的一些常用方法及其详细介绍：

1. lock()

- **功能**：尝试将 std::weak_ptr 转换为 std::shared_ptr。如果 std::weak_ptr 所指向的对象仍然存在（即引用计数不为 0），则返回一个指向该对象的 std::shared_ptr，同时对象的引用计数加 1；如果对象已经被销毁，则返回一个空的 std::shared_ptr。
- **示例代码**：

```
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
    std::weak_ptr<int> weakPtr = sharedPtr;

    if (auto newSharedPtr = weakPtr.lock()) {
        std::cout << "Object is still alive. Value: " << *newSharedPtr <<
std::endl;
    } else {
        std::cout << "Object has been deleted." << std::endl;
    }

    sharedPtr.reset();
    if (auto newSharedPtr = weakPtr.lock()) {
        std::cout << "Object is still alive. Value: " << *newSharedPtr <<
std::endl;
    } else {
        std::cout << "Object has been deleted." << std::endl;
    }

    return 0;
}
```

2. expired()

- **功能**：检查 `std::weak_ptr` 所指向的对象是否已经被销毁。如果对象已经被销毁（即引用计数为 0），则返回 `true`；否则返回 `false`。
- **示例代码**：

```cpp
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
    std::weak_ptr<int> weakPtr = sharedPtr;

    if (weakPtr.expired()) {
        std::cout << "Object has been deleted." << std::endl;
    } else {
        std::cout << "Object is still alive." << std::endl;
    }

    sharedPtr.reset();
    if (weakPtr.expired()) {
        std::cout << "Object has been deleted." << std::endl;
    } else {
        std::cout << "Object is still alive." << std::endl;
    }

    return 0;
}
```

3. `reset()`

- **功能**：将 `std::weak_ptr` 置为空，即不再指向任何对象。这不会影响所指向对象的引用计数。
- **示例代码**：

```cpp
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
    std::weak_ptr<int> weakPtr = sharedPtr;

    weakPtr.reset();
    if (weakPtr.expired()) {
        std::cout << "Weak pointer is now empty." << std::endl;
    }

    return 0;
}
```

4. use_count()

- **功能**：返回与 std::weak_ptr 共享同一对象的 std::shared_ptr 的数量。注意，std::weak_ptr 本身不影响引用计数，因此这个值反映的是实际拥有对象所有权的 std::shared_ptr 的数量。
- **示例代码**：

```
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sharedPtr1 = std::make_shared<int>(42);
    std::shared_ptr<int> sharedPtr2 = sharedPtr1;
    std::weak_ptr<int> weakPtr = sharedPtr1;

    std::cout << "Number of shared pointers: " << weakPtr.use_count() <<
std::endl;

    return 0;
}
```

5. 赋值操作

- **功能**：可以使用 std::shared_ptr 或另一个 std::weak_ptr 对 std::weak_ptr 进行赋值操作，使其指向相同的对象。
- **示例代码**：

```
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
    std::weak_ptr<int> weakPtr1;
    std::weak_ptr<int> weakPtr2;

    weakPtr1 = sharedPtr;
    weakPtr2 = weakPtr1;

    if (auto newSharedPtr = weakPtr2.lock()) {
        std::cout << "Value: " << *newSharedPtr << std::endl;
    }

    return 0;
}
```

这些方法使得 std::weak_ptr 能够在不影响对象生命周期的情况下，安全地观察和操作对象，是解决循环引用问题的重要工具。

new 和 malloc 的区别

在 C++ 中，`new` 和 `malloc` 都用于动态分配内存，但它们在功能和行为上有显著区别：

1. **内存分配方式**：
   - `malloc`：是 C 语言中的标准库函数，用于从堆区分配指定大小的内存块。使用时需要显式指定所需内存的字节数，并返回 `void*` 类型的指针，需要进行类型转换。

     ```
     int* ptr = (int*)malloc(sizeof(int));
     ```

   - `new`：是 C++ 中的运算符，用于从自由存储区（通常与堆相同）分配内存。`new` 会根据所请求的类型自动计算所需的内存大小，并返回该类型的指针，无需显式类型转换。

     ```
     int* ptr = new int;
     ```

2. **类型安全性**：
   - `malloc`：返回 `void*` 类型的指针，需要显式类型转换，可能导致类型不安全。
   - `new`：直接返回所请求类型的指针，确保类型安全。

3. **内存初始化**：
   - `malloc`：仅分配内存，不初始化所分配的内存内容。
   - `new`：在分配内存的同时，调用对象的构造函数进行初始化。

4. **内存分配失败处理**：
   - `malloc`：如果分配失败，返回 `NULL`。
   - `new`：如果分配失败，抛出 `std::bad_alloc` 异常。

5. **内存释放方式**：
   - `malloc`：使用 `free` 函数释放内存。

     ```
     free(ptr);
     ```

   - `new`：使用 `delete` 运算符释放内存。

     ```
     delete ptr;
     ```

6. **构造和析构函数调用**：

- `malloc`：不会调用对象的构造函数，释放内存时也不会调用析构函数。
- `new`：分配内存时会调用构造函数，释放内存时会调用析构函数。

综上所述，`new` 是 C++ 提供的内存分配运算符，除了分配内存外，还负责对象的初始化和类型安全，符合 C++ 的面向对象特性。而 `malloc` 是 C 语言的内存分配函数，主要用于分配内存，不涉及对象的构造和析构。在 C++ 中，推荐使用 `new` 和 `delete` 来进行动态内存管理，以充分利用 C++ 的特性。

delete 和 free 有什么区别?

在 C++ 中，`delete` 和 `free` 都用于释放动态分配的内存，但它们有显著的区别，主要体现在以下方面：

1. **适用对象不同**：
   - `free`：用于释放通过 `malloc`、`calloc` 或 `realloc` 等函数分配的内存，这些函数在 C 语言中用于动态内存分配。
   - `delete`：用于释放通过 `new` 或 `new[]` 运算符分配的内存，这些运算符在 C++ 中用于动态内存分配。
2. **内存释放过程**：
   - `free`：仅释放内存，不会调用对象的析构函数，因此如果对象包含需要释放的资源（如动态分配的内存），可能导致资源泄漏。
   - `delete`：在释放内存之前，会先调用对象的析构函数，确保对象的资源得到正确释放。
3. **语法和类型安全**：
   - `free`：需要包含头文件 `<cstdlib>`，并且返回类型为 `void*`，使用时需要进行类型转换，可能导致类型不安全。
   - `delete`：是 C++ 的运算符，使用时不需要头文件支持，且具有类型安全性，直接与对象类型匹配。
4. **与数组的配对使用**：
   - `free`：释放单个对象或数组时，都使用 `free`，但需要确保释放的内存块与分配时的大小一致。
   - `delete`：释放单个对象时使用 `delete`，释放对象数组时使用 `delete[]`，以确保正确调用每个对象的析构函数。

示例：

使用 `malloc` 和 `free`：

```
#include <cstdlib> // 包含 malloc 和 free 的头文件

class Obj {
public:
    Obj() { /* 构造函数 */ }
    ~Obj() { /* 析构函数 */ }
    // 其他成员函数
```

```
};

void useMallocFree() {
    Obj* obj = (Obj*)malloc(sizeof(Obj)); // 使用 malloc 分配内存
    if (obj != nullptr) {
        // 使用 obj
        free(obj); // 使用 free 释放内存
    }
}
```

使用 `new` 和 `delete`：

```
class Obj {
public:
    Obj() { /* 构造函数 */ }
    ~Obj() { /* 析构函数 */ }
    // 其他成员函数
};

void useNewDelete() {
    Obj* obj = new Obj(); // 使用 new 分配内存并调用构造函数
    // 使用 obj
    delete obj; // 使用 delete 释放内存并调用析构函数
}
```

总结：

- 在 C++ 中，建议使用 `new` 和 `delete` 来进行动态内存管理，以确保对象的构造和析构过程得到正确处理。
- `malloc` 和 `free` 属于 C 语言的标准库函数，在 C++ 中使用时需要注意类型转换和手动调用构造/析构函数。
- 切勿将 `new` 和 `free`，或 `malloc` 和 `delete` 混合使用，以避免未定义行为。

什么是野指针,怎么产生的,如何避免

在 C++ 编程中，**野指针**（Wild Pointer）是指未被初始化或已指向无效内存区域的指针。使用野指针进行解引用操作可能导致程序崩溃、数据损坏或其他未定义行为。

野指针的产生原因：

1. **未初始化的指针：**
    - 在声明指针变量时，如果未对其进行初始化，它将指向一个随机内存地址。此时，指针所指向的内存内容无法预测，使用该指针可能导致意外结果。

```
int* ptr; // 声明指针但未初始化
*ptr = 10; // 未定义行为，可能导致程序崩溃
```

2. **释放内存后未将指针置空：**
   - 在使用 `delete` 或 `free` 释放指针所指向的内存后，如果不将指针设置为 `nullptr`（在 C++11 及以上版本）或 `NULL`，该指针将成为悬空指针（Dangling Pointer）。再次使用该指针可能访问已释放的内存，导致不可预测的行为。

```
int* ptr = new int(10);
delete ptr; // 释放内存
// ptr = nullptr; // 应将指针置为 nullptr
*ptr = 20; // 未定义行为，可能导致程序崩溃
```

3. **指针超出作用域：**
   - 当指针指向的对象超出其作用域（例如，函数结束时），如果指针仍然存在并尝试访问该对象，将导致野指针。

```
int* ptr;
{
    int num = 10;
    ptr = &num;
} // num 的作用域结束，ptr 成为野指针
*ptr = 20; // 未定义行为，可能导致程序崩溃
```

避免野指针的方法：

1. **初始化指针：**
   - 在声明指针时，将其初始化为 `nullptr` 或 `NULL`，确保指针在使用前指向有效地址。

```
int* ptr = nullptr; // C++11 及以上版本
// 或
int* ptr = NULL; // C 语言风格
```

2. **释放内存后置空指针：**
   - 在使用 `delete` 或 `free` 释放内存后，立即将指针设置为 `nullptr` 或 `NULL`，防止悬空指针的产生。

```
int* ptr = new int(10);
delete ptr;
```

```
ptr = nullptr; // 防止悬空指针
```

3. **避免返回指向局部变量的指针：**
   - 不要返回指向局部变量的指针，因为局部变量在函数结束时会被销毁，指针将指向无效内存。

```
int* func() {
    int num = 10;
    return &num; // 错误，返回指向局部变量的指针
}
```

4. **使用智能指针：**
   - 在 C++ 中，使用智能指针（如 `std::unique_ptr`、`std::shared_ptr`）来管理动态内存，智能指针会在超出作用域时自动释放内存，减少手动管理内存的错误。

```
std::unique_ptr<int> ptr = std::make_unique<int>(10);
// ptr 超出作用域时，内存会自动释放
```

通过遵循上述方法，可以有效避免野指针问题，提高程序的安全性和稳定性。

野指针和悬浮指针的区别

在 C++ 中，**野指针（Wild Pointer）** 和 **悬挂指针（Dangling Pointer）** 是两种常见的指针误用问题，它们都可能导致程序崩溃或未定义行为，但成因和场景不同。以下是它们的核心区别：

---

1. 野指针（Wild Pointer）

- **定义**：指针变量 **未被初始化**，指向一个 **随机的、未知的内存地址。**
- **成因**：
  - 指针声明后未赋值（例如 `int* ptr;`）。
  - 指针被直接赋值为一个非法地址（例如 `int* ptr = 0x12345678;`）。
- **风险**：
  - 访问野指针（如 `*ptr = 10;`）可能覆盖任意内存，导致程序崩溃、数据损坏或安全漏洞。
- **示例**：

```
int* ptr;        // 野指针（未初始化）
*ptr = 42;       // 未定义行为：可能崩溃或静默破坏内存
```

---

2. 悬挂指针（Dangling Pointer）

- **定义**：指针指向的 **内存已被释放**，但指针未被置空，仍然保留着原来的地址。
- **成因**：
  - 动态内存被释放后未置空指针（例如 `delete ptr;` 后未设置 `ptr = nullptr;`）。
  - 指向局部变量的指针在函数返回后继续被使用（如返回局部变量的地址）。
- **风险**：
  - 访问已释放的内存（如 `*ptr = 10;`）可能导致 **段错误（Segmentation Fault）** 或读取到无效数据。
- **示例**：

```
int* func() {
    int x = 10;
    return &x;     // 返回局部变量 x 的地址（x 在函数返回后被销毁）
}

int* ptr = func(); // ptr 成为悬挂指针
*ptr = 20;         // 未定义行为：x 的内存已被回收
```

核心区别总结

特性	野指针	悬挂指针
初始化状态	**未初始化**，指向随机地址	**已初始化**，但指向已释放的内存
触发场景	未赋值的指针被访问	内存释放后指针未置空
风险类型	覆盖未知内存，导致不可预测行为	访问无效内存，导致崩溃或数据错误
修复方法	初始化时赋值为 `nullptr` 或有效地址	释放内存后立即置空指针

如何避免？

1. **野指针**：
   - **始终初始化指针**：

```
int* ptr = nullptr;  // 明确初始化为空
```

   - 使用 **智能指针**（如 `std::unique_ptr`、`std::shared_ptr`）替代裸指针。
2. **悬挂指针**：
   - 释放内存后立即置空指针：

```
delete ptr;
ptr = nullptr;  // 避免悬挂
```

- **避免返回局部变量的地址或引用。**
- 使用智能指针的 `reset()` 方法自动管理内存生命周期。

代码示例对比

```
// 野指针示例
int* wild_ptr;         // 未初始化，指向随机地址
// *wild_ptr = 10;     // 危险：可能破坏内存！

// 悬挂指针示例
int* dangling_ptr = new int(42);
delete dangling_ptr;  // 内存释放
// *dangling_ptr = 50; // 危险：访问已释放内存！
dangling_ptr = nullptr; // 修复：置空指针
```

内存对齐的概念和作用，为什么需要考虑内存对齐？

内存对齐是指数据在内存中的存储地址必须满足特定对齐条件，通常要求变量的首地址是其类型大小的整数倍。例如，4字节的int类型变量必须存储在4的倍数的地址上。这种机制由编译器自动实现，但程序员也可通过`#pragma pack`等指令干预。

为什么需要考虑内存对齐？

内存对齐的核心原因包括**硬件效率优化**和**硬件兼容性**，具体分析如下：

---

1. 硬件效率优化：减少CPU访问内存次数

现代CPU以固定块（如4字节或8字节）读取内存。若数据未对齐，CPU需要多次访问内存并拼接数据，显著降低性能。例如：

- **对齐场景**：int变量存储在地址0x0000，CPU一次读取即可获取全部4字节。
- **非对齐场景**：int变量存储在地址0x0001，CPU需读取0x0000-0x0003和0x0004-0x0007两个块，再剔除冗余字节，才能获取全部4字节。

场景	CPU访问次数	性能影响
内存对齐	1次	高效，无额外操作
内存未对齐	2次	需拼接数据，效率下降50%

---

2. 硬件兼容性：避免程序崩溃

部分处理器（如ARM、MIPS）不支持非对齐内存访问，尝试读取未对齐数据会触发硬件异常。例如：

- **Alpha处理器**：直接拒绝访问未对齐数据，导致程序崩溃。
- **x86架构**：虽支持非对齐访问，但性能下降明显（约损失2-3倍速度）。

---

3. 缓存利用率提升

内存对齐可减少**缓存行（Cache Line）浪费**。例如：

- **缓存行大小**：通常为64字节。
- **非对齐结构体**：若跨两个缓存行，需加载两次数据，增加缓存未命中率。
- **对齐优化**：确保数据集中在单个缓存行内，提升缓存局部性（如表2）。

结构体布局	缓存行占用	性能影响
非对齐（跨两个行）	2行	缓存命中率低，访问延迟高
对齐（单行内）	1行	缓存命中率高，访问速度快

---

4. 实际开发中的影响案例

- **SIMD指令集（如AVX）**：要求数据32字节对齐，否则无法执行或引发崩溃。
- **多线程伪共享（False Sharing）**：若不同线程的变量位于同一缓存行，频繁写入会导致缓存行无效化，降低并发性能。通过内存对齐隔离变量可避免此问题。

内存对齐的规则与优化

1. **结构体对齐规则**：
    - 成员偏移量为其大小与编译器对齐值中较小者的整数倍。
    - 结构体总大小需为最大成员对齐值的整数倍。

```
struct Test {
    char a;    // 1字节，偏移0
    int b;     // 4字节，偏移4（1+3填充）
    short c;   // 2字节，偏移8（4+4）
};             // 总大小12字节（需补齐至4的倍数）
```

2. **优化建议**：
    - **调整成员顺序**：将大尺寸变量（如double）放在前面，减少填充字节（如表3）。
    - **编译器指令**：使用`#pragma pack(n)`调整对齐系数（需权衡性能与内存占用）。

结构体成员顺序	总大小（字节）	填充字节数
char, int, short	12	5
int, char, short	8	1

总结

内存对齐通过**减少CPU访问次数**、**避免硬件异常**和**提升缓存效率**，成为高性能编程的关键技术。开发中需结合硬件特性与编译器规则，合理设计数据结构，必要时通过指令微调对齐策略，以平衡性能与内存开销。